CHAPTER 2

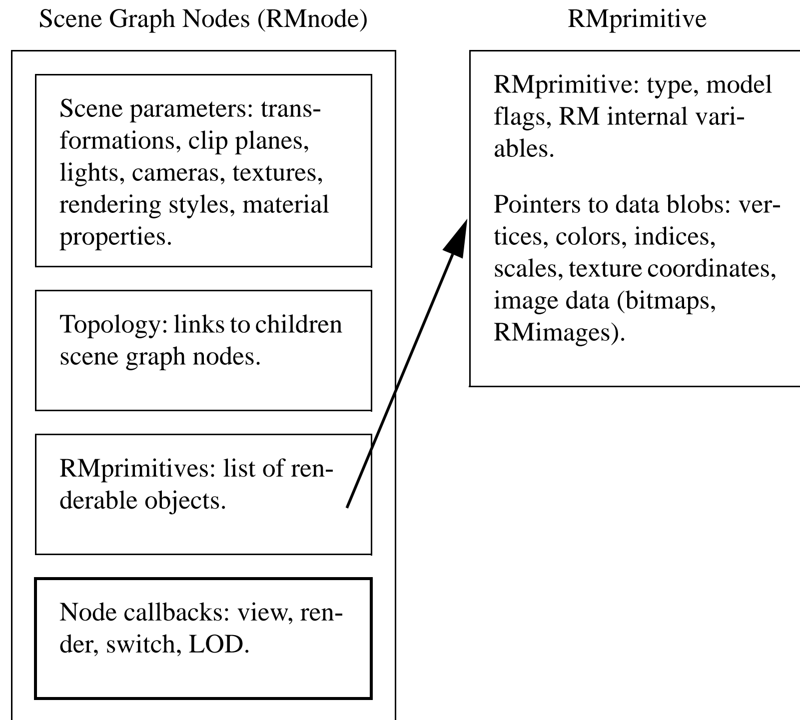
RM Scene Graph Theory of Operation

This chapter provides an overview of the entire workings of RM Scene Graph. In relation to the rest of the manual, this chapter serves as a roadmap to the entire RM system, including forward references to concepts as they are presented. The intent is to give a brief but technical overview of key components of RM technology. Each of the topics presented in this technical overview is covered in detail later in this manual.

The Scene Graph

From a topological perspective, the scene graph is a collection of nodes and edges. The nodes contain renderable data (what is drawn) or rendering parameters (how it appears). The edges define relationships between the nodes in the scene graph. There is only one type of node in the RM scene graph: the *RMnode*. The RMnode is initially a compact structure that may grow to contain renderable objects called *primitives*, rendering parameters called *scene parameters*, and optional application callbacks called *node callbacks*. Renderable geometry in *RMprimitives* is drawn only in scene graph leaf nodes (those nodes with no children).

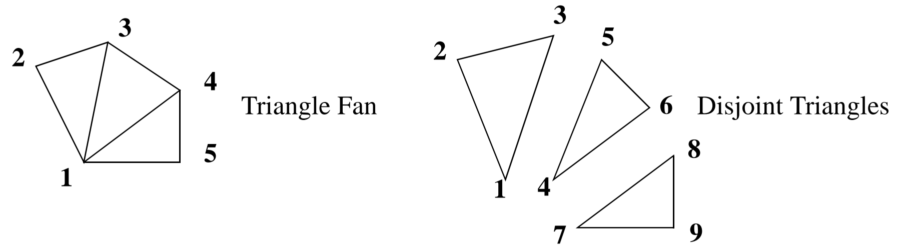
FIGURE 3. RM Scene Graph Nodes and Primitives



Renderable Primitives Overview

A renderable object in the RMnode is called an *RMprimitive*. Renderable objects fall into two classes, structured and unstructured. Unstructured primitives are simply a list of disjoint objects. In contrast, the objects in a structured primitive obey a set of rules defining an implicit relationship between the components of the primitive. An example of a structured primitive is a triangle fan, while an example of an unstructured primitive is a list of disjoint triangles.

FIGURE 4. Structured and Unstructured Primitives



The triangle fan uses an ordering rule that describes *N-2* triangles from *N* vertices. There is an implicit relationship between triangles in this primitive. Each *N*'th vertex in the fan uses the previous vertex and the first vertex to define a triangle. In contrast, there is no relationship between the triangles in the disjoint triangles primitive.

Each RMnode in the scene graph may contain an arbitrary number of RMprimitives. RMprimitives assigned to leaf nodes in the scene graph will be drawn during the rendering pass; RMprimitives assigned to interior scene graph nodes are silently ignored and will not be drawn. Each RMprimitive may contain a single structured primitive or a single unstructured primitive. Each of the structured or unstructured primitives may be of arbitrary size. In other words, each RMprimitive may be thought of as an array of objects of a single type. An RMprimitive may define an arbitrary number of disjoint triangles, or may define a single triangle fan consisting of an arbitrary number of triangles. The collection of RMprimitives at an RMnode may be of different types. A single RMnode may contain both a triangle fan and a disjoint triangle primitive. The only restriction on the RMprimitives owned by an RMnode is that they must all be composed of 2D or 3D vertices. It is not possible to mix 2D and 3D primitives in a single node. We'll elaborate on this issue in the section later in this chapter entitled "Frame-Based Rendering" on page 53. In addition to RMprimitives, an RMnode may also contain a list of children RMnodes. More precisely, the list of children is a list of pointers to RMnodes.

Scene Parameters Overview

Whereas primitives define what is rendered, scene parameters define how the primitives appear when rendered. Scene parameters include things like the position of a viewer world space, the presence or absence of clipping planes, location and properties of light sources, viewport geometry. A list of all RM scene parameters appears later in this chapter, and in complete detail in *Chapter 9, Scene Parameters.*

Node Callbacks Overview

RM supports a rich set of node-level callbacks that are under application control. Applications may set, or unset, node-level callbacks that are invoked during one of the two stages of the rendering traversal. Although callbacks are typically used to invoke application-supplied code, RM provides pre-built callbacks to implement common tasks like view frustum culling. Callback entry points are provided for invocation during either of the two stages of the rendering traversal to provide fine-grained control to the application at key points during the rendering process.

When each callback is invoked, RM passes several parameters to the callback routine. One of these parameters is a *render state* object. The render state object contains information about the rendering state at the time the node is encountered. Rendering state information includes things like model and view transformation matrices, window width, current line width and shading model, and so forth.

The *pretraversal* callback is invoked when the traverser first begins processing of an RMnode. The pretraversal callback is invoked prior to processing of any other RMnode parameters. This callback can produce premature traversal termination depending upon the value returned. The primary purpose of this callback is to provide applications a way to terminate the traversal of the scene graph rooted at the node containing the callback. A common use of this callback is to perform view frustum culling, thereby terminating traversal at a scene graph node that is not visible.

The second type of callback is called a *switch* callback. After the scene parameters at an RMnode have been processed, but before any children nodes are processed, the switch callback is invoked. When an RMnode has no switch callbacks, all the children of the RMnode will be traversed at render time. The presence of the switch callback implies that one and only one child (one subtree) of the RMnode will be processed at render time. The switch callback returns an index that selects which child of the RMnode will be processed.

The third type of callback is the *post traversal* callback. Unlike the pretraversal or the switch callback, the post traversal callback does not cause the traversal order to be modified. The post traversal callback is invoked after the RMnode and any children RMnodes have been processed.

Finally, RM provides a unique type of callback we refer to as a *render order* callback. Generally speaking, children of an RMnode are processed in the order they were added to the parent by the application. In some instances, it may be necessary

for the application to alter the order in which the children are processed, as well as to alter the number of children processed. The render order callback will allow the application to specify the index order in which children of a given RMnode will be processed. The render order callback is a powerful operator that can be used to implement advanced rendering capabilities, like sorted transparency.

Data Model

The term *data model* refers to how data is stored and manipulated in a system. RM implements a data model that allows for shared data management with applications. Without shared data management, all data presented to the scene graph must be copied. As the size of the scene graph grows, shared data management becomes a necessity. Shared data management is one way to avoid making multiple copies of data.

RM addresses the shared data management issue by dividing scene graph data into two classes: *metadata* and *data blobs*. Data blobs are large homogenous collections of data. An array of object vertices, normals or colors is a data blob. The raw pixel data used as a texture map is a data blob. On the other hand, metadata is most often descriptive information, containing information such as the size of an array, the name of an object, or the number of data blobs owned by an object.

The RM data model is straightforward but remarkably effective: all metadata is copied, while all data blobs may or may not be copied. When management of data blobs is shared, the application must also provide a callback used to free blob resources when no longer needed by RM. Also, when data blobs are shared, there is only one copy of the blob present in the system. When blob management is not shared, RM makes a copy of the data blob. Each approach has benefits and drawbacks.

Data blobs appear in RM as large blocks of homogeneous data in primitives and image objects. RM stores “bulk” data in data blobs: vertices, colors, normals, texture coordinates, image pixel data and so forth. RM offers the ability to share data management of such bulk data with the application. This capability means that RM need not make a separate copy of the bulk data, thereby minimizing memory consumption. To achieve a minimum memory footprint, applications should instruct RM to use immediate rather than retained mode rendering so that a copy of the bulk data is not cached into a display list.

RM Primitive Types

RM provides a rich collection of primitive types. Surface primitives describe geometric surfaces in three dimensions, or shapes in two dimensions. Vector primitives are composed of line segments. Semi-procedural primitives can be thought of as recipes describing more elaborate shapes or surfaces using relatively few parameters. Image-based primitives contain two- or three-dimensional blocks of pixel data. A platform-neutral text primitive is also provided. This section is intended to provide an overview of the available primitives in RM Scene Graph. Further details about each primitive, including the APIs, follow in subsequent chapters.

Each RMprimitive is comprised of one or more types of bulk data provided by the application. Some bulk data types are required, while others are optional. For instance, triangle-based primitives require vertex geometry. Optional triangle data consists of surface normals, colors, and texture coordinates.

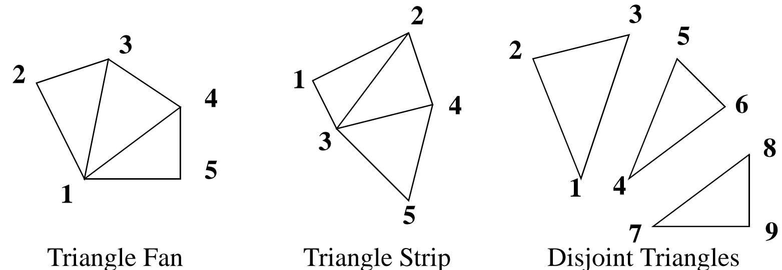
Explicit Surface and Shape Primitives

Triangle Primitives

The disjoint triangle primitive is an unstructured collection of independent triangles. Each triangle is independent of every other triangle. A total of *N* vertices defines *N/3* triangles. All triangle primitives must have vertex information, but surface normals, colors and texture coordinates are optional. Disjoint triangles may have surface normal presented by the application in either per-vertex or per-triangle format.

The triangle strip and triangle fan primitives define structured groups of triangles where each triangle shares vertices and edges with others in the primitive. Each triangle in the strip or fan shares an edge (two vertices) with the previous and next triangle in the strip or fan. Applications must specify vertex data for the triangle primitive. Optional data – surface normals, texture coordinates and colors – is specified on a per-vertex basis for both the triangle strip and fan, but may be specified on a per-face or per-vertex basis for the disjoint triangles primitive.

FIGURE 5. Triangle Primitives



Quadrilateral Primitives

RM defines three primitive types that are based upon quadrilaterals. They are the quadmesh, quad strip and disjoint quads primitive types. All of these primitives types require that vertex data be present. Optional data – surface normals, texture coordinates and colors – must be specified on a per-vertex basis for quad strip and quad mesh primitives, but may be specified on either a per-vertex or per-face basis for disjoint quads. Quad-based primitives are often a convenient way to represent surfaces. Since they may be non-planar, developers should be aware of their limitations with regard to rendering and display issues. The quadmesh and quad strip RMprimitives are structured, while the disjoint quads RMprimitive is unstructured.

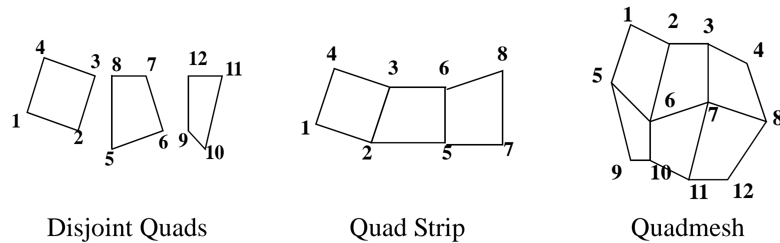
The disjoint quads primitives is used to define *N/4* disjoint quadrilaterals from a total of *N* vertices. With disjoint quads, applications may specify surface normals on a per-vertex or per-face basis. You may specify an arbitrary number of disjoint quadrilaterals in a single RMprimitive.

The quad strip primitive is used to define a total of *N-2* quadrilaterals from *N* vertices. The first and final quad in the strip share an edge and two vertices with adjacent quads in the strip. Interior quads share an edge and two vertices with both the previous and next quad in the strip. You may specify a single quad strip of arbitrary length with this RMprimitive.

The quadmesh primitive is an extension of the quad strip object. Whereas the quad strip defined a single strip of connected quads, the quad mesh compactly represents a two-dimensional array of quad strips. The vertex ordering is slightly different than that used for quad strips. The vertex array for quadmesh primitives can be considered as a lattice of points defined over a range of two parametric dimensions. You may specify a single quadmesh primitive of arbitrary size with this RMprimi-

tive. The figure below illustrates the vertex ordering differences between the quadrilateral primitive types.

FIGURE 6. Quads, Quad Strip and Quadmesh and Primitives



Vector Primitives

There are two vector based primitives in RM, the polyline strip (structured) and disjoint lines (unstructured). In a polyline strip, each *N* vertices defines a total of *N-1* line segments. Each segment in the polyline strip shares a vertex with the previous and next line segment. In contrast, the disjoint line primitive uses *N* vertices to define *N/2* disjoint line segments. In disjoint line segment primitives, there is no vertex sharing. All line segment primitives require vertex data. Optional data – surface normals (yes, they can be assigned to line segments to induce shading), texture coordinates and colors – are assigned on a per-vertex basis for the polyline strip, or on a per-vertex or per-segment basis for the disjoint lines primitive.

Semi-Procedural Primitives

Some primitives, such as triangles, require a complete geometric description. The triangle requires the application to specify each of the three triangle vertices in order to fully define the triangle. In contrast, the surfaces of some primitives may be implicitly described by an equation. A sphere is a good example of such a primitive. All spheres are objects that are defined by the formula $R^2 = X^2 + Y^2 + Z^2$. A unique sphere can be defined with just a center point and a radius, rather than requiring applications to tessellate the sphere into a large number of triangles or other surface primitives.

This section describes a number of semi-procedural primitives. All of the semi-procedural primitives described in this section may be described in a compact way, and RM uses the compact description to generate a more elaborate representation.

As of the time of this writing, there is no mechanism in RM to assign texture coordinates to semi-procedural primitives (spheres, cones, cylinders). It is possible to use a pre- and post-traversal callback on the RMnode containing the spheres primitive to enable/disable automatic OpenGL texture coordinate generation. The RM demonstration program *autoTmap.c* uses exactly this technique to implement textured spheres.

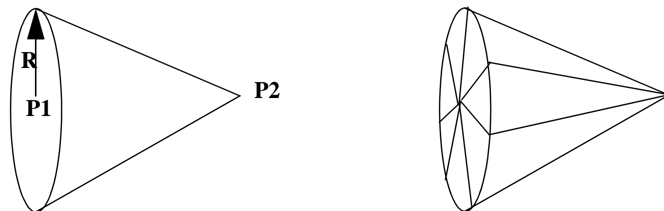
Spheres

The RM spheres primitive contains a list of sphere centers and radii. This primitive may contain an arbitrary number of spheres. The application specifies the location of each sphere with a single vertex that corresponds to the sphere's center point in world coordinates. Optional data – colors and radius values – are specified on a per-sphere basis. In the event an application does not specify radius values, RM assigns a default radius value of 0.0 to the sphere. Depending upon the characteristics of the underlying OpenGL implementation, the resulting degenerate triangles may or may be visible.

Cones

This RM primitive contains a list of data that specifies cone geometries. This primitive may contain an arbitrary number of cones. As shown in Figure 7, cone geometry is described with three parameters; two vertex values and a radius value. The two vertices define a line segment that is the center of the cone, and a single radius value defines the width at the base of the cone. The first vertex is the center point of the cone base, while the second vertex is the position of the cone apex. A total of *N* vertices defines *N/2* cones. Color and radius values are optional, but are specified on a per-cone basis. Radius values are specified on a per-cone basis. Like the sphere primitive, applications have direct control over how finely the cone surface is tessellated via the RMprimitive model flag. The cone body and bottom are both tessellated to the same resolution using triangles as shown below in Figure 7.

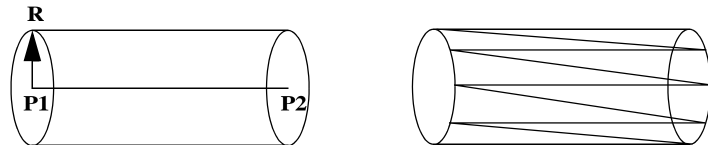
FIGURE 7. RM Cones Primitive



Cylinders

Like spheres and cones, the RM cylinders primitive represents an unstructured list of cylinders. Each cylinder in this primitive shares no information with any other cylinder. Each cylinder is specified with two vertices, so a cylinders primitive that contains *N* vertices defines *N/2* cylinders. Optional radius and color data is specified on a per-cylinder basis. Cylinder surface tesselation may be controlled using the primitive's multiresolution modeling parameter. Figure 8 shows how cylinders are tessellated. Unlike cones, the cylinder is not "capped" on the ends; it is a hollow tube with open ends.

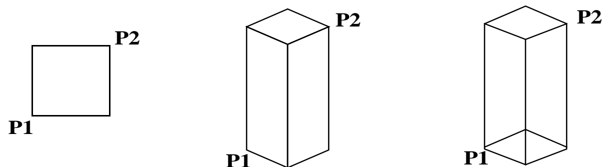
FIGURE 8. RM Cylinders Primitive



Boxes

RM offers three different types of box primitives. One is used to represent 2D boxes. The other two are used to represent 3D boxes, with one form dedicated to solid boxes and the other to wireframe boxes. Note that it is possible to manipulate the polygon rendering mode of nodes containing 3D solid boxes so they appear drawn as wireframe. Each box primitive uses two vertices to specify the corners of the box, both in 2D and 3D. Optional color data is specified on a per-box basis. Surface normals are generated automatically by RM for the 3D, filled box primitive. Each box is drawn so that is axis-aligned with the world coordinate system. Transformation scene parameters may be applied at an RMnode to transform all boxes within an RMprimitive.

FIGURE 9. RM Box Primitives

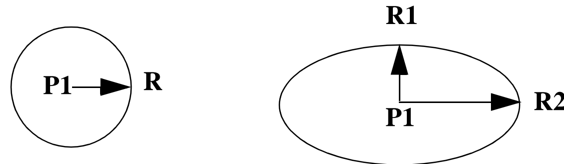


Circles and Ellipses

In RM, circles and ellipses are 2D objects – RM does not have a primitive for a 3D circle or ellipse. Both circles and ellipses are specified with a single vertex, defining the center point. Circles use a single radius value per vertex (per center point definition), while ellipses use two radius values per vertex. While vertex data is required, radius data is optional for both circles and ellipses. In the event an application does not specify radius values, an default radius value of 0.0 is supplied by the RM draw code. The resulting degenerate polygon may or may not be visible depending upon the OpenGL implementation. Optional data for circles and ellipses consists of per-vertex color.

Internal to RM, the circle and ellipse primitives are tessellated at render time into *n*-gons, where *n* is computed at render time to produce minimal aliasing. Applications may use the RMnode polygon rendering mode scene parameter to control whether the primitives are drawn in outline or filled mode. Similarly, applications may alter the line width and dashing styles via node-level scene parameters. Ellipses are drawn axis-aligned. Like the boxes primitive, transformation of ellipses is achieved by applying a transformation matrix to the node owning the ellipse primitive (or to an ancestor node).

FIGURE 10. RM Circles and Ellipses



Octmesh

The RM octmesh is a procedural primitive used for volume rendering. The octmesh is a 3D analog to the quadmesh primitive. The application specifies resolution values corresponding to the octmesh’s width, height and depth along with two 3D coordinates that provide the extents of the octmesh “defining volume.” At render time, RM will draw a number of axis-aligned quadrilaterals in back-to-front view order. An RMprimitive level model flag may be manipulated to adjust the render-time resolution of the octmesh geometry.

By itself, the octmesh primitive is a compact description of some procedural geometry like a sphere or circle. To implement volume rendering, applications must acti-

vate 3D texture mapping by assigning a 3D texture as a scene parameter at a node that either owns the octmesh RMprimitive, or at an ancestor node.For traditional direct volume rendering, applications should also schedule the RMnode containing the octmesh primitive for rendering during the 3D transparent pass. Additionally, the application-supplied 3D texture should be a four channel, RGBA volume.

FIGURE 11. RM Octmesh Hardware-Accelerated Volume Rendering.

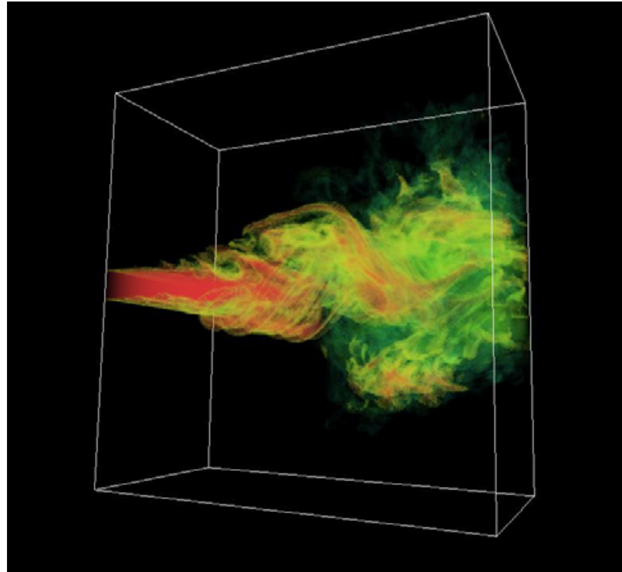


Image and Bitmap Primitives

RM provides two types of image primitives: sprites and bitmaps. A "sprite" is an image that is positioned in space. It is often used as a replacement for complex geometry to achieve better rendering performance in games and interactive animations. The sprite image can be any of the RMimage types supported by RM. In contrast, the bitmap primitive's "image data" may consist only of RMbitmap image objects.

Both sprite and bitmap primitives exhibit the behavior and constraints of OpenGL-based pixel drawing operations. In both 2D and 3D, OpenGL renders the primitive's image data parallel to the view plane and axis-aligned with the edges of the display window. Similarly, there is no perspective foreshortening effect on sprite or bitmap images. Applying a transformation to a sprite or bitmap-based primitive

will transform only the position of the primitive. In other words, a transformation moves the lower left corner of the primitive, and the sprite or bitmap is still drawn axis-aligned with the edges of the display window. If your application requires rotatable images, you should use texture mapping.

All sprite and bitmap-based primitives are subject to pixel zoom operations, which are specified as node-level scene parameters in RM. In addition, when the lower left corner of the bitmap lies outside the window boundaries, the entire bitmap will be culled. The sprite primitive uses draw code designed to prevent such behavior, and that code will be extended to bitmap primitives in a future RM version.

Sprite

The required data for sprite primitives consists of vertex and RMimage data. The *n*-th vertex value positions the lower left corner of the *n*-th RMimage object in 2D or 3D. There are no optional data fields. Sprites are not subject to the effects of lighting, but sprites positioned and rendered in 3D are subject to fogging effects.

Bitmap

The required data for bitmap primitives consists of vertex and RMbitmap data. Bitmap primitives require that there be the same number of vertex values and RMbitmaps. Unlike sprite primitives, which derive image pixel color from the RMimage object, the bitmap primitive is drawn using the “current color.” That means you may specify a color at the RMnode level that will be used for all “on pixels” in the bitmap. “Off pixels” are not drawn. Alternatively, you may specify per-vertex color to the bitmap primitive – color data is optional for bitmap primitives – and all “on bits” will be set to the per-vertex color.

Text Primitives

From the developer’s perspective, the text primitive in RM provides a platform-neutral implementation of 2D text. RM implements text by creating a cache of bitmap glyphs for each character of a font family and style specified using the text properties scene parameter. Because text is implemented using OpenGL bitmaps, text follows the same set of rules as bitmaps: text may be positioned with 2D or 3D vertices, and is always rendered parallel to the view plane, and always rendered aligned with the edges of the display window. Required data for the text primitive consists of vertex data and text strings.

Indexed Primitives

In the preceding discussions, the number of triangles or quads drawn for each of the different primitive types is a function of the number of vertices contained in the primitive. For example, specifying a triangle strip or triangle fan using a total of *N* vertices will result in a total of *N-2* triangles being drawn. The point here is that the total number of triangles drawn at render time is a function of the number of vertices your application provides when building the RMprimitive. The same concept generalizes to other primitive types: quads, lines, etc.

Indexed primitives are a bit different in that the number of triangles or quads drawn at render time is a function of the number of *index values* contained in the primitive rather than the number of vertices. As with non-indexed primitives, your application will populate the indexed primitive with required and optional data. Indexed primitives introduce an additional required field: an array of index values. The index values array consists of integers that are used as indices into the arrays of other RMprimitive data: vertices, normals and colors. Since the number of index values determines the number of renderable "things" (triangles, bitmaps, etc.), the number of "things" drawn for each of the indexed primitive types is a function of the number of indices. For each *i*'th "thing" drawn, RM uses the *i*'th index to access a data element across all data blobs within the primitive.

Not all primitives have indexed variants. As of the time of this writing (August 2005), the following primitive types have indexed variants: bitmap, text, triangles, triangle strips, triangle fans, quads, and quad strips.

Multiresolution Control

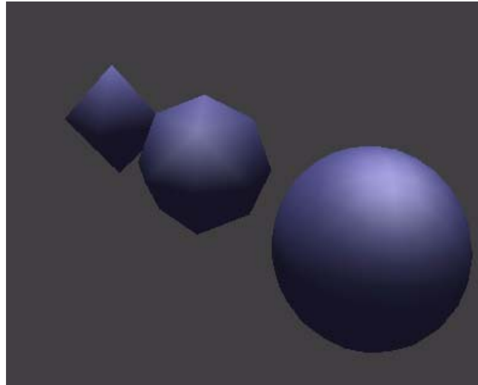
The term "multiresolution control" refers to the process of selecting one of many possible representations of an object at render time. Multiresolution control is an effective way to improve rendering performance by reducing load on the graphics pipe. Sometimes known as "level of detail" control (LOD), the basic idea behind multiresolution control is to have multiple representations of a single renderable object in the graphics database. The multiple representations span a range of resolution, from coarse to fine. When the object is close to the viewer, the high resolution model is rendered, but when the object is far away from the viewer, the lower resolution model is rendered. The premise is rendering high resolution details into a few pixels is a waste of resources, and that a low resolution depiction is visually adequate when the model is far from the viewer. RM provides two forms of multiresolution control: static and dynamic.

Static multiresolution control is provided for certain types of primitives (spheres, cylinders, cones and the octmesh) in the form of a “model flag” attribute that is set at the RMprimitive level. The model flag selects one of several pre-tessellated models, regardless of distance to the viewer.

In contrast, dynamic multiresolution control is implemented programatically through by using a switch callback attached to an RMnode. During the view traversal, RM invokes the switch callback and provides information so the switch callback can compute the distance from the viewpoint to the center of the bounding box of the scene graph node. Based upon the distance, the switch callback will return an index that selects one of several children nodes that will traversed.

Figure 12 shows three sphere models of varying resolution. The sphere closest to the viewer is rendered at the maximum tessellation level provided by RM, while the sphere furthest from the viewer is rendered using the most coarse tessellation sphere model.

FIGURE 12. Switch Callbacks are often used to improve rendering performance by selecting lower-resolution models farther from the viewer at render time to reduce load on the graphics pipeline.



Scene Parameters

RM scene parameters specify “how” objects in the scene appear when rendered. They are assigned at the RMnode level and have a scope of control that extends only to children nodes, but not sibling nodes.

The scene parameters can be divided into the following categories:

1. Placement of viewports within the display window.
2. Control over how the framebuffer is cleared within a viewport.
3. Specification of viewer position.
4. Light source and light model control.
5. Material properties, including color, along with rendering style.
6. Textures and texturing environments.
7. Clipping planes.
8. Text properties.

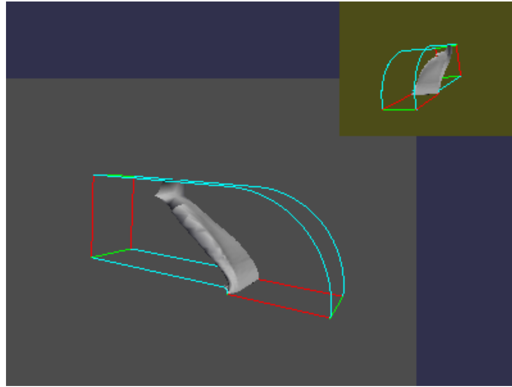
Viewports

The viewport scene parameter allows applications to specify a region within the display window into which rendering occurs. When you specify a viewport scene parameter, all rendering generated at children nodes will be constrained to the viewport. You may use multiple viewports within the scene graph to produce "more interesting" renderings.

In Figure 13, we used a total of three viewports in a single scene graph. The top-level viewport occupies the entire window. At a scene graph node one level deeper than the root node, we specified a viewport that occupies the lower-left corner of the display window. At a sibling node at the same level, another viewport contains the image in the upper-right corner of the window.

While the viewport scene parameter manages display window real estate, we used a different camera and background clear color in each of the viewports so that different views of a scene appear in each of the viewports. The viewport itself just specifies a rectangular region within the current display window where subsequent rendering will occur. Viewports are specified in RM in *normalized device coordinate* (NDC) space so that the application need not be concerned with the details of window size. Depending upon how you design your scene graph, it is possible to use a single geometric transformation to change the position of geometry rendered in both viewports.

FIGURE 13. **Multiple Viewports**



Framebuffer Clear and Fill Modes

Framebuffer clears in RM are accomplished by specifying a background fill operator and assigning it as a scene parameter to either an RMpipe or an RMnode. Background fill operators may be a single color or an image. Background fill operations consist of either a single value or an RMimage, and the fill operation may be applied to either the color or depth planes of the framebuffer. A “single value” framebuffer clear operator will cause the entire region within the currently defined viewport to be set to the single value. In contrast, an image framebuffer clear operator will be tiled across the viewport.

Developers may assign the background fill operator at either the RMpipe level or the RMnode level. Assigning the fill operator at the RMpipe level will cause the framebuffer (and depth buffer) clear operations to be executed prior to any rendering when you invoke the frame-based renderer using *rmFrame()*. In contrast, when the background fill operator is assigned at a scene graph node, the operator is not executed until that node is traversed during rendering. In most instances, developers will assign the background fill operator to the RMpipe when only one framebuffer “clear” is needed per frame. The example shown in Figure 11 above uses a total of three different clear operators, one for each of the three visible viewports.

When assigning background fill operators at the RMnode level, developers must take care to ensure that nodes containing background clear scene parameters are executed only once during multipass rendering to avoid the most common RM programming error: buffer clears erase the rendering results of previous passes in multipass rendering.

Note that framebuffer and depth buffer clears are not a default operation in RM. This means you must explicitly enable framebuffer and depth buffer clears by specifying background clear operators as scene parameters at either the RMpipe level (executed once per frame) or at the RMnode level (possibly executed multiple times per frame depending upon the node's traversal mask).

Background Color and Depth Value

In the simplest case, a single value can be assigned as a scene parameter to an RMnode in order to activate framebuffer or depth buffer clears. In the case of a framebuffer clear, the single value is a four-component color vector (RMcolor4D). In the case of the depth buffer, the single value is a floating point scalar in the range 0.0 to 1.0.

When a background color value is assigned as a scene parameter at an RMnode, if no depth buffer clear value exists for that node, then one is automatically created. This means that if you assign a background clear color, the depth buffer will also be cleared "for free," without the need for an addition depth buffer clear scene parameter assignment.

Background Color and Depth Images

The alternative to a single-valued clear operation is to use an image that is copied or tiled into the color or depth buffer. Use of background color or depth images is a handy way to present complex backgrounds without the overhead of additional rendering. For example, you could pre-render a complex static scene, then read back the depth and color buffers, and use the resulting images to initialize the color and depth buffers in subsequent frames. Then, in each subsequent frame, you could rerender only the dynamic portion of the scene into the complex background. This approach can produce dramatic increases in rendering speed in some situations.

One of the RM demonstration programs, *pdb*, uses exactly this technique to accelerate rendering of molecular bonds as line segments into a pre-rendered scene containing atomic positions represented as spheres.

If the image you provide for either a color or depth buffer background clear operation does not exactly match the size of the display window, the image will be tiled across the window. The upper left corner of the source image will coincide with the upper left corner of the display window. The right and bottom edges of the display window will be "ragged" with respect to the source image.

Like the single-valued background color planes clear scene parameter, if you specify an image to use for clearing the color planes, the depth buffer will be cleared for you automatically to a single value. You can override this behavior by assigning NULL as the depth clear value; this will effectively disable initialization of the depth buffer during rendering, but will not disable the depth test during 3D rendering passes.

Cameras and Viewpoints

The location and orientation of the viewer is specified with a 2D or 3D camera scene parameter. The view specification defines a transformation that has scope of control at all children scene graph nodes. 2D cameras should be used only for 2D objects, and 3D cameras should be used only for 3D objects.

2D Cameras

The 2D camera scene parameter defines a rectangular region in world coordinates. Objects that are positioned inside this window will be visible in the viewport, while objects that lie outside the area visible by the 2D camera will be clipped away. The view area specified by the two, 2D coordinates is scaled along width by the aspect ratio parameter of the 2D camera scene parameter. Adjusting the aspect ratio will cause objects to appear “squished” or “stretched.” This adjustment is useful for preserving proportion of object appearance when the area specified by the camera is not square.

3D Cameras

Whereas the 2D camera defines a viewing region, the 3D camera defines a viewing volume in the shape of a truncated right pyramid. The position of the viewer, the eye coordinate, is a 3D vertex specified in world coordinates. The viewer is looking in the direction of the “at point,” which is also a 3D vertex specified in world coordinates. A 3D “up vector” specified the world coordinate direction that will map to the “up direction” when the view area is projected into the 2D viewport. These parameter, the eye coordinate, the look-at coordinate and the up vector define the position and orientation of the viewer.

The size of the viewing volume, or frustum, is specified with four additional scalar values. The “field of view” parameter specifies the angle of the viewing volume along the projected Y-axis. The horizontal angle of the viewing volume is computed from the product of the field of view, and the aspect ratio. These two parameters, aspect ratio and field of view, define the shape of the viewing volume. The

location of the near and far clipping planes are specified as positive distances from the eye.

3D cameras implement either perspective or othorgraphic projection, which is specified with a single enumerator.

Stereoscopic Cameras

A stereoscopic camera in RM is a 3D camera, as described above, but which makes use of additional parameters that are used to implement binocular stereo viewing geometry. Two parameters, one defining the focal length, and one defining an angle for the degrees of separation between the eyes, are used to compute offset vectors at render time. These offset vectors modify the location of the eye point to produce new eye points for each of the left and right eyes.

Lights and Lighting Models

Objects in the 3D world are illuminated by light sources, which may be individually controlled and placed. RM implements control of light sources as scene parameters. The result is that the illumination effects of any given light source in the scene graph extends only to descendent scene graph nodes. The RM illumination controls are intended to provide access to all of OpenGL's illumination parameters, but using a simplified and compact interface.

Light Sources

RM implements eight individually controllable light sources[1]. Three light source types are supported: point light source, directional light and a spotlight. Like OpenGL light sources, the RM light sources use color vectors for each of the diffuse, ambient and specular reflection components. Spotlights have additional parameters that control the spotlight dispersion angle, and the light intensity falloff characteristics.

1. All OpenGL implementations support up to eight light sources, while some implementations support more. RM supports only eight light sources, which correspond to the eight supported by all OpenGL implementations. It is not possible to gain access to any additional light sources through the RM interface, even if supported by the underlying OpenGL implementation.

Light Models

The RM light model scene graph parameter allows the application to specify a global ambient illumination term that is applied in addition to the ambient reflectance term computed during vertex lighting. In addition, the light model controls whether or not lighting is "two-sided." Finally, the light model implements an interface to OpenGL's "local viewer" parameter. The local viewer parameter alters how the lighting equation is evaluated. In one case, local viewer enabled, the shade at each vertex is computed using the direction vector to the position of the eyepoint as specified using the 3D camera scene parameter. In the other case, local viewer disabled, the shading equation is evaluated as if the viewer is located at the origin of the world coordinate system. Such shading can be performed more quickly, but at the expense of visual quality.

Material Properties and Rendering Styles

RM implements numerous related scene parameters that affect the appearance of objects. Material properties affect the shading equation by providing separate colors for each of the ambient, diffuse and specular components of the object. Other types of rendering styles include line width and dashing styles, polygon outline mode, and control over how front- and back-facing polygons are rendered.

Lambertian Surface Reflectance Model

The shading model used by RM and OpenGL follows the Lambertian reflectance model: $Color = Ambient + Diffuse + Specular$. In this equation, the *Ambient* term is the product of the ambient component of the light source with the coefficient of ambient reflectance of the object; the *Diffuse* term is the dot product of the surface normal at the vertex with the direction to the light source multiplied by the diffuse reflectance coefficient of the object; and the *Specular* term is the dot product of the vector to the viewer and the reflected light vector, raised to a power known as the specular exponent, then multiplied by the specular reflectance coefficient of the object. The OpenGL lighting model adds a twist to this basic shading equation by allowing the developer to specify a color vector, rather than a single coefficient, for each of the ambient, diffuse and specular terms for both the light source and the object itself. OpenGL also supports a transparency component in the color vector, so that transparency can be modified on a per-vertex basis as well as made a function of the shading equation, if desired.

In the relatively common case where the color of the object changes on a per-vertex basis, RM uses OpenGL's color tracking (glColorMaterial) to change the object's

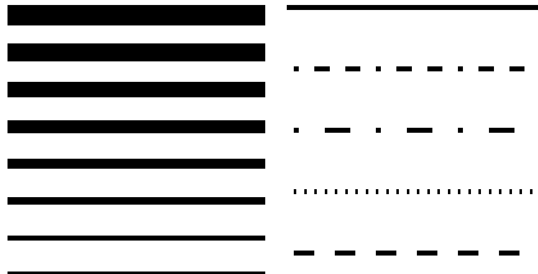
diffuse and ambient reflectance colors to match the color of each vertex. The color of a specular reflection is the color of the light source causing the specular reflection. Multiple specular reflections from light sources of different colors produce an additive effect.

Line Widths and Dashing Styles

The style of line segments is manipulated through the use of node-level scene parameters that control the width and dashing style of line segments. When you change one or both of these scene parameters at a scene graph node, all line segments in children nodes are rendered using the new style.

To specify a new line width or dashing style, you will use one of the predefined widths or styles using an enumerator. RM provides eight line width enumerators, and five dashing styles. The underlying code base is engineered such that a developer may extend the number of width or style enumerators.

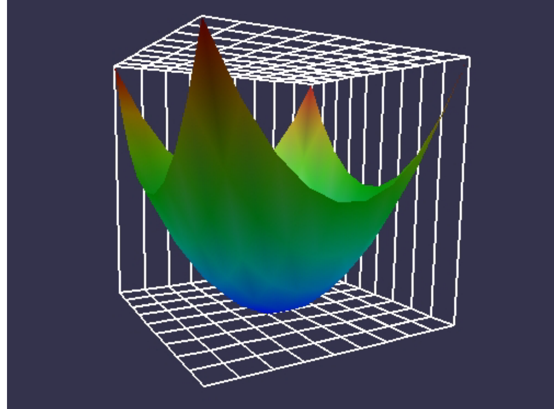
FIGURE 14. Line Widths and Dashing Styles



Polygon Rendering Modes

During the rendering process, polygon vertices are first transformed, then a color for each vertex is computed using current material and lighting properties, then the vertices are passed along to a rasterizer. Polygon rendering modes control the appearance of polygons at the rasterization stage.

FIGURE 15. Polygon fill and cull modes. The grid in the scene consists of polygons that surround the central object. They are rendered using outline mode. Backfacing polygons are culled so that the interior is visible. This image was produced by the *vis3d.c* demonstration application.

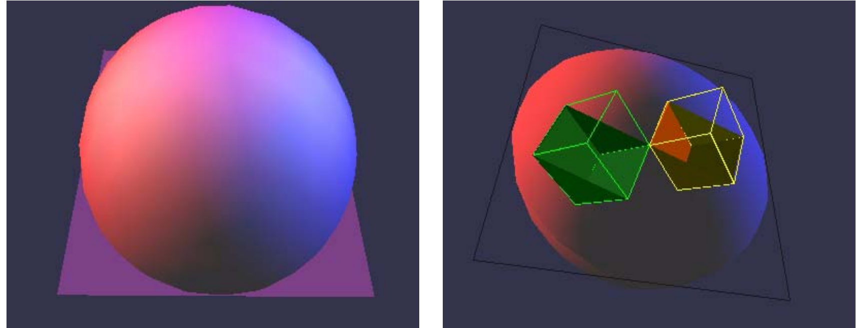


When rasterized, polygons may be rendered such that the interiors are filled (the default mode), or such that only edges or the vertices themselves are visible. In addition, it is possible to completely disable rendering of front- or back-facing polygons. The image above illustrates a combination of polygon edge rendering combined with rejection of back facing polygons: the outline grid (in white) consists of a number of polygons that are rendered using a combination of outline-only and backface culling.

Clipping Planes

Clipping planes in RM are implemented as a scene parameter at the node level. When active, a clip plane will be active and affect rendering of geometry at all children nodes. Clip planes are a useful means to create “cutaway” views of objects.

FIGURE 16. The *clipper.c* demonstration application illustrates how to interactively transform a clipping plane.



Up to six clip planes may be specified at any given scene graph node. Each may be individually enabled or disabled. A clip plane is specified using a point-normal formulation in world coordinates. Clip planes are affected by geometric transformations present in the scene graph at ancestor nodes.

Textures, Texture Mapping and Environments, Multitexturing

Texture mapping is the process of "pasting" an image onto a geometry[2] in order to increase visual detail without increasing geometric resolution. Generally speaking, the texture mapping process requires an image along with "texture coordinates" at each geometry vertex. The texture coordinates are treated as parametric indices into the texture, and define how the image pixels map onto the polygon.

In order to activate texture mapping in RM, you must specify an RMtexture object as a texturing scene parameter, and your RMprimitive must have texture coordinates. The RMtexture object contains both texture pixel data as well as texturing environment attributes. Your application specifies the texture pixel data by assigning one or more RMimage objects to the RMtexture. Multiple RMimage objects correspond to the usual OpenGL mipmaps. After being populated with texture pixel data, your application assigns the RMtexture object as a scene parameter to a scene graph node, and then texture mapping is then active at all children scene graph nodes.

Applications must use one of the *rmPrimitiveSetTexcoord* routines to assign texture coordinates to RMprimitives. Generally speaking, there must be a one-to-one corre-

2. Technically, textures are applied at the pixel fragment level during rasterization.

spondence between the number of texture coordinates and geometry vertices. In other words, only per-vertex texture coordinate data is supported.

Textures in RM may be 1D or 2D images, or 3D volumes. The figure below shows several textures in the same scene, each of which is applied to a different polygon.

FIGURE 17. The *tcube.c* demonstration application uses a different texture on each of the six faces of a cube.

Whereas the texture coordinates define a parametric index into a texture map from a pixel fragment, the texture environment defines how the texture elements are combined with the existing pixel fragment shade to produce a final pixel color. The texturing environment in RM implements an interface to most of OpenGL's texturing environment parameters. Texture elements may be combined with computed polygon shade using one of the following modes: decal, modulate, replace or blend. Refer to the OpenGL Red Book for more information about each of these modes.

Multitexturing refers to the process of combining two or more textures at each pixel fragment to produce a final color. RM's multitexturing implementation builds upon the unitexturing model where one or more RMtextures are assigned as scene parameters to an RMnode using *rmNodeSetSceneMultiTexture*. At the RMprimitive level, each set of texture coordinates per multitexture is specified using one of the *rmPrimitiveSetMultiTexcoord* family of routines.

Geometric Transformations

Geometric transformations in RM are implemented as a set of related scene parameters. Transformations alter the position and orientation of objects in the scene. As scene parameters, they have affect over an entire rooted subtree of the scene graph. Depth-wise accumulation of geometric transformations is performed automatically by RM. The order of operations causes transformations at deeper levels to be applied before transformations at more shallow levels.

The composite geometric transformation at a scene graph node is computed using the following equation: $Pre \cdot -C \cdot S1 \cdot R \cdot S2 \cdot C \cdot T \cdot Post$. These terms are defined as:

TABLE 1. Geometric Transformation Components

Pre	An arbitrary 4x4 matrix.
-C	A 3-element vector defined by the node's center point.
S1	A "pre-rotation" scale matrix.
R	A rotation matrix.
S2	A "post-rotation" scale matrix.
C	The 3-element vector defining the node's center point.
T	A 3-element vector used for translation.
Post	An arbitrary 4x4 matrix.

This set of transformation components provides a great deal of flexibility in specifying arbitrary geometric transformations. Transformations may be applied to either texture coordinates, or to geometry, which includes primitive vertices, lights, cameras and clip planes.

Text Properties

The text properties scene parameter controls the appearance of text primitives in the scene graph. The text properties object specifies the font family, size and text style that is used when rendering text primitives. RM implements five font family enumerators, and seven font size enumerators. A pair of boolean values in the text properties object control font italicization and emboldening. Render-time justification of text strings is specified in the text properties scene parameters, so that text strings can be left, right or center justified.

Window Systems, Rendering Pipes and Initialization

One of the most challenging aspects of scene graph programming is the process of initializing all resources needed in order to begin rendering. Application initialization is the confluence of three related ideas:

- Creating and initializing a window for use in the windowing system;
- Opening and initializing an OpenGL rendering context, and
- Initializing any additional structures required by the scene graph system.

RM Scene Graph applications will perform several steps in order to initialize an *RMpipe* object. The RMpipe is the control structure that holds information specific to a single rendering thread. This information includes a window handle, an OpenGL context, and internal structures that are used to control how rendering is performed.

There are two broad approaches that can be used by RM applications in order to perform RMpipe initialization. The first approach is to use several related RM functions that create a window and that cause the RMpipe to become initialized for use in rendering. An alternative, which is preferred in some circumstances, is for the application to create the window for rendering, to create an OpenGL rendering context and to assign these values to the RMpipe using the appropriate RM calls. Both approaches are equally correct, and there is no particular advantage in one over the other.

TABLE 2. RMpipe Initialization (Terse).

Activity	How Accomplished
Create a new RMpipe.	rmPipeNew()
Assign a channel format (mono, anaglyph stereo or multibuffered stereo).	rmPipeSetChannelFormat()
Assign a processing mode to the RMpipe (see the section on multithreaded rendering, below). The default processing mode is RM_PIPE_MULTISTAGE.	rmPipeSetProcessingMode()
Create a window for rendering.	rmauxCreateW32Window() for Win32 systems, or rmauxCreateXWindow() for X11.

TABLE 2. RMpipe Initialization (Terse).

Activity	How Accomplished
Assign the window to the RMpipe.	rmPipeSetWindow()
Make the RMpipe "current", activating the OpenGL context for use in rendering.	rmPipeMakeCurrent()

Once initialized, the RMpipe includes references to an OpenGL-capable window into which objects are rendered, and an active OpenGL rendering context. Once these steps are completed, the application may begin to invoke the RM frame-based renderer, which is described in more detail in a subsequent section

As a general rule, applications should create and initialize an RMpipe object prior to beginning scene graph construction. This is not a requirement, but there are some RM routines that assume that an OpenGL rendering context has been created and is ready for use. For example, RM provides a routine that will use the graphics hardware to build mipmaps. This action can dramatically accelerate the process of building mipmaps for 3D textures.

Event Management

Once the RMpipe has been initialized, applications may begin to construct a scene graph and to invoke the frame-based renderer. Interactive applications can be built using the RMaux library, which contains routines that create action handlers associated with window system events. A default set of action handlers is provided in RMaux that implements interactive transformation of objects and cameras, as well as response to keyboard, window and mouse events. Applications can create their own set of action handlers to implement a desired user interface paradigm.

In addition to event action handlers, the RMaux library provides a one-time callback referred to as the "initialization function[3]."

RMaux also provides an idle callback, which is invoked when there are no pending events in the queue (X11) or after a finite amount of time has elapsed (Win32).

3. The RMaux initialization function callback is different from and should not be confused with the RM initialization function *rmInit*.

Alternatively, applications are free to implement their own event management system. RM has been used extensively with other systems that implement an event handling system, such as the Fast Light Toolkit (FLTK)[4], the GIMP Toolkit (GTK)[5] and the CAVE library[6].

Frame-Based Rendering

Multipass and Stereo Rendering

RM uses a multipass rendering architecture to implement a flexible framework for supporting a variety of rendering tasks, such as anaglyph or multibuffered stereo and 2D-over-3D layering. The enumeration and order of rendering passes is as follows: 3D, opaque objects are rendered first, followed by 3D transparent objects, then 2D objects. Any of these three passes may be selectively enabled or disabled by changing an RMpipe's attributes.

Stereo rendering modes replace a single rendering traversal with left- and right-channel traversals. The 3D camera scene parameter has attributes that will enable automatic computation of left- and right-eye binocular viewing geometry that is based upon a monoscopic viewpoint. RM supports three types of stereo display modes: red-blue anaglyph, blue-red anaglyph, or multibuffered. The anaglyph modes will work on any platform that supports OpenGL, while the multibuffered mode works only on hardware that supports quad-buffered stereo.

Node Traversal Masks

Scene graph nodes contain three types of "traversal" masks. Generally speaking, a traversal mask is a bit field that specifies whether or not a given RMnode is processed during a scene graph traversal.

The first type of traversal mask "schedules" a node for processing during one or more passes of the RM multipass rendering traversal. That traversal mask is created and assigned at the time the RMnode is created using *rmNodeNew()*. Furthermore,

4. Fast Light Toolkit, http://www.fltk.org/
5. GIMP Toolkit, http://www.gtk.org/
6. CAVELibrary, http://www.vrco.com/

that traversal mask cannot be changed by the application once the node has been created. The RMnode may be scheduled for processing during one or more of the three RM rendering passes: 3D, opaque; 3D, transparent; 2D.

The second type of traversal mask is referred to as the "traverse enable" flag. This traversal mask is used to control whether or not an RMnode is processed during any of the rendering passes. Applications may manipulate or obtain this RMnode attribute using *rmNodeSetTraverseEnable* or *rmNodeGetTraverseEnable*. When RMnodes are created, the "traverse enable" flag is set to RM_TRUE. That default value may be overridden by calling *rmSetEnum*.

The third type is used during pick operations conducted when calling *rmFramePick* or *rmFramePickList.* In some cases, applications may wish to constrain pick operations to a subset of scene graph nodes in order to improve performance or to implement a certain type of user interface action. When RMnodes are created, the pick traversal mask is set to RM_TRUE. This default setting may be overridden by calling *rmSetEnum* to assign a new default pick traversal mask value that is assigned by *rmNodeNew*. Therefore, to exclude an RMnode from a pick operation, explicit application action is required. The pick traversal attribute may be manipulated by calling *rmNodeSetPickEnable* or obtained using *rmNodeGetPickEnable*.

Applications may exclude a rooted subtree of the scene graph from processing during rendering or pick operations simply by manipulating the traverse enable or pick enable flags at the root node of the subtree. During the depth-first, left-to-right traversal, when an RMnode's traverse enable flag (during rendering) or pick enable flag (during picking) is set to RM_FALSE, no children nodes will be processed.

Frame Callbacks

Applications may attach callbacks to the RMpipe object, and these callbacks are described in the table below. The RMpipe callbacks are associated with activities at the conclusion of frame-based rendering.

TABLE 3. RMpipe Frame Callbacks

Name	Description
rmPipeSetPostRenderFunc	Upon conclusion of rendering, RM will create an RMimage object containing the color planes of the framebuffer. This image will be provided to the application callback. This callback is typically used by applications to create raster files.
rmPipeSetPostRenderDepthFunc	Instead of providing an RMimage of the color planes of the framebuffer, RM will create an RMimage containing the contents of the depth buffer. The depth buffer image is provided to the application callback.
rmPipeSetPostRenderBarrierFunc	The "barrier callback" is invoked after the image callbacks, but before the swap buffers callback. This callback can by used by multithreaded applications to force synchronization across multiple rendering pipes.
rmPipeSetSwapbuffersFunc	By default, RM sets this RMpipe method to one that will swap buffers. Applications can override this behavior, possibly setting the RMpipe swap buffers function to NULL so that third party event management software can control the buffer swap.

Picking

RM uses the OpenGL selection and feedback mechanisms in order to implement object picking. Picking is invoked as a frame-based rendering subroutine call, and will return either the closest object at the pick location, or a list of all objects encountered at the pick location. The pick operation returns an RMpick record (or multiple RMpick records, depending upon which of the two RM picking operators is used), which the application may query to obtain information about the picked object.

Constant-Rate Rendering

With a single call, you can have RM render frames at a constant rate of speed. Many applications, like virtual environments, require that frames be rendered and delivered at a constant rate of speed. When your application requests a certain frame rate, RM will take steps to ensure that no more than the requested number of frames per second are generated and displayed. This approach works well when the rendering time for a given scene is less than the requested frame rate. This approach does not help in the case where the rendering cost for a complex scene exceeds the amount of time allotted per frame in a constant-rate rendering scheme. In that case, it is really the application's responsibility to reduce rendering load through model simplification. Providing the ability for applications to obtain a measurement of rendering load and to respond accordingly will be the subject of future work.

Postscript Output

RM Scene Graph (but not OpenRM) provides a frame-based rendering method that will create a PostScript file from the scene graph. The PostScript file is a true, vector representation, as opposed to a PostScript file containing a raster image. The RM PostScript file is suitable for use on high-resolution printers, and is especially useful for imaging operations that require full device resolution to prevent aliasing of vectors and text.

Multiprocessing and Thread Safety

The RM scene graph is designed to be "thread safe." This means that your application can create multiple rendering threads (multiple RMpipes) that simultaneously render into multiple windows from a single scene graph. This type of architecture is needed in order to perform high-performance, hardware-accelerated rendering into multiple windows of a tiled projection system (Powerwall), or in a surround-style immersive Virtual Reality system (a CAVE). In addition, your application may consist of multiple threads that simultaneously write into the scene graph. This architecture is useful for balancing input/output operations with rendering.

In addition to being thread safe, RM intrinsically supports multistage and multi-threaded rendering. A multistage decomposition of the rendering pipeline produces two rendering "processes". The first, called the *view traversal*, is where all view-dependent operations are performed. Objects can be culled against the view frustum, so that objects not visible are not passed along to the graphics pipeline for ren-

dering. Distance-based level-of-detail model selection is also performed during the view traversal. The second stage, called the *render traversal*, is where objects produced by the view traversal are dispatched to the graphics hardware for rendering.

This multistage architecture can be run serially, or with each stage running in parallel. Applications can select between serial or parallel traversals by manipulating the "processing mode" attribute of an RMpipe. The following table lists the processing modes supported by the RMpipe.

TABLE 4. RMpipe Processing Modes

Mode	Description
RM_PIPE_MULTISTAGE	The view and render traversals are executed sequentially and in the same thread.
RM_PIPE_MULTISTAGE_PARALLEL	The view and render traversals are each placed into a separate execution thread, and both are performed simultaneously (parallel).
RM_PIPE_MULTISTAGE_VIEW_PARALLEL	The view traversal is placed into a separate execution thread, while the render thread remains in the same thread as the caller. Both traversals are executed simultaneously (in parallel).
RM_PIPE_SERIAL	(Deprecated 1.6.0) The view and render traversals are combined into a single scene graph traversal.

RM provides two types of parallel rendering in order to solve a fundamental problem that arises when combining a scene graph system with a third-party toolkit that performs event management. The fundamental problem is that some third-party software makes the assumption that it is the steward of the OpenGL rendering context. The RMpipe mode RM_PIPE_MULTISTAGE_PARALLEL will attempt to assign the OpenGL rendering context to a detached rendering thread. This may cause catastrophic failure in some third-party software that assumes ownership of the context.

Distributed Memory Parallel Rendering

RM Scene Graph supports a form of distributed memory parallel rendering, which can be used to perform rendering on a cluster of machines equipped with graphics hardware. Such a configuration is common in environments with segmented, high

resolution display murals composed of individual projectors. For such an environment, RM makes explicit use of the capabilities of Chromium[7], which is stream-based framework that intercepts OpenGL calls from an application and routes them to multiple graphics servers.

Any RM application can be used in a one-to-many form of parallelism, where a single RM application drives many displays. In addition, RM provides capabilities that allow a developer to construct a distributed memory parallel application in which many application Processing Elements (PEs) can simultaneously drive many graphics servers. Our reference implementation for a distributed memory parallel application uses MPI, but RM is sufficiently flexible to support *any* type of parallel programming model you choose. The details of the implementation are described in *Chapter 16, Parallel Rendering with RM and Chromium*as well as in several technical publications downloadable from the r3vis.com website.

Interfacing RM with Third-Party Tools

RM has been designed and implemented in such a way as to be very flexible and easy to integrate with third-party software tools.

Third Party Modeling Applications

Some third-party modeling applications such as Cal3D are capable of generating complex surface models and textures that can then subsequently be used in OpenGL applications. RM provides the means for your application to assign OpenGL display lists created by third-party tools at the RMprimitive level using *rmPrimitiveSetAppDisplayList*. Similarly, a third-party texture identifier can be assigned to an RMtexture using *rmTextureSetTextureID* in place of the texel data normally specified using RMimages. The real benefit here is that your application can leverage the benefits of both the scene graph infrastructure as well as the capability of third-party modeling tools to create robust, visually rich applications.

Third Party GUI and Event Processing Frameworks

RM provides the means for your application to perform any or all of the duties of window management, event processing and OpenGL context management. A spe-

7. See http://chromium.sourceforge.net/

cial RMpipe enumerator - RM_PIPE_NOPLATFORM - may be used with the *rmPipeNew* call to indicate that your application will perform all window and OpenGL context management, and will perform the swapbuffers call outside of RM's frame-based rendering routine. Applications that use third-party GUI and event processing frameworks, like FLTK or Qt, will find it very easy to integrate RM's scene graph capabilities.

Summary

RM is a robust and rich framework for implementing high performance graphics applications. The RM Scene Graph implementation reduces developer time by encapsulating many OpenGL details behind a consistent and straightforward API. This chapter presents a brief overview of RM's capabilities, data objects and theory of operations. The rest of this Programming Guide explores each of these topics in detail, providing reference material invaluable to the RM developer.